Lecture 27:

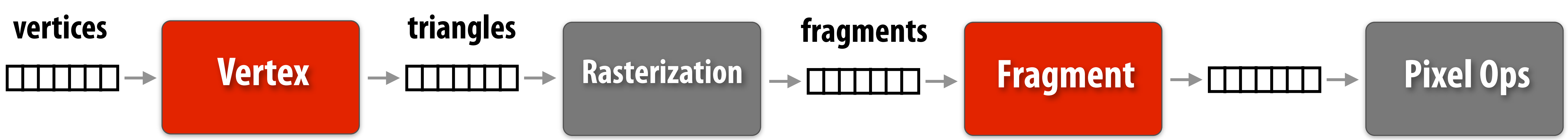
Flexible Graphics Pipelines

(programmable global structure, not just programmable stages)

Visual Computing Systems

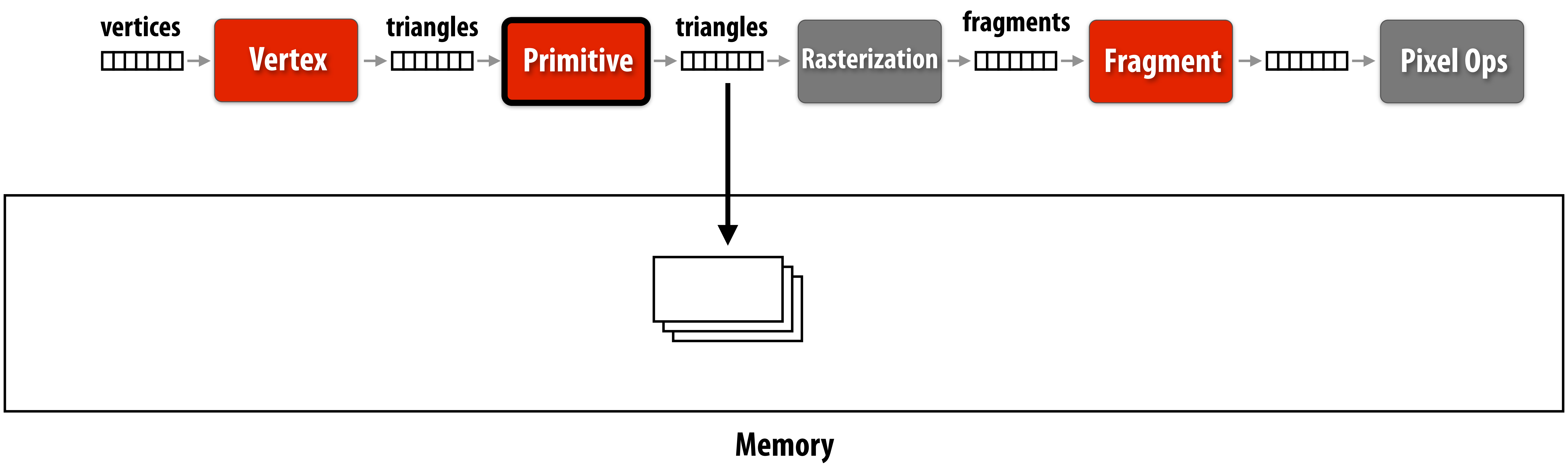
CMU 15-869, Fall 2013

Graphics pipeline pre Direct3D 10



Graphics pipeline circa 2007

[Blythe, Direct3D 10]

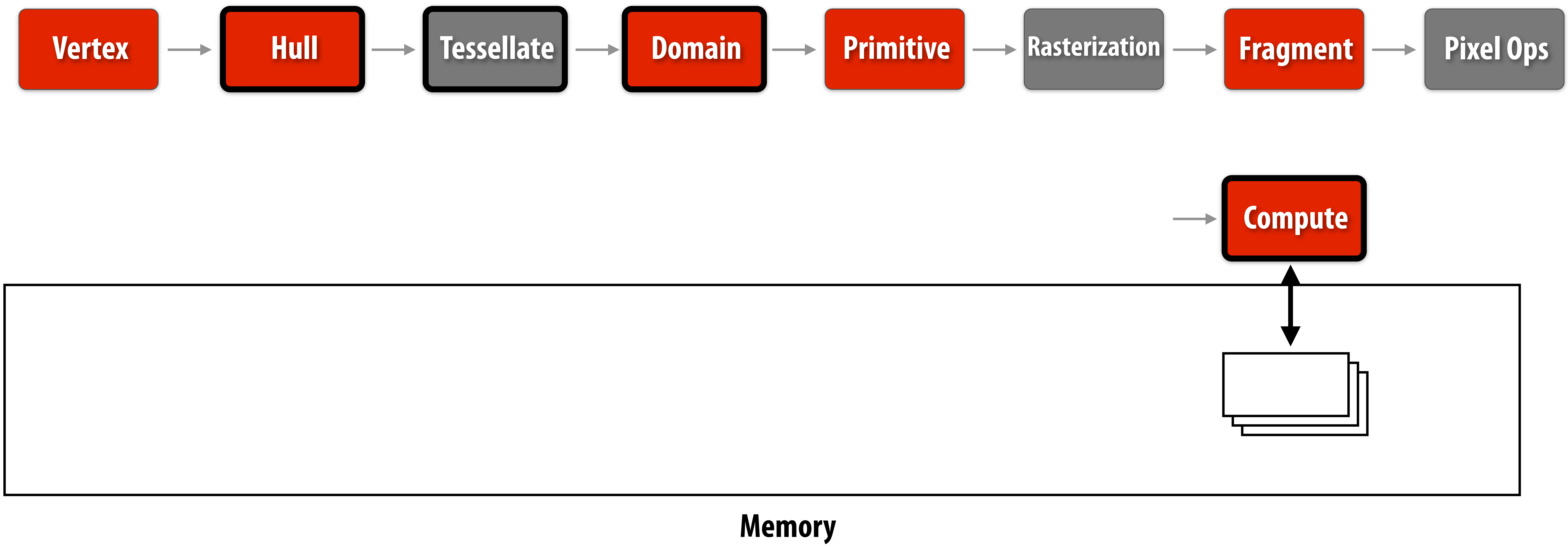


Added new stage

Added ability to dump intermediate results out to memory for reuse

Pipeline circa 2010

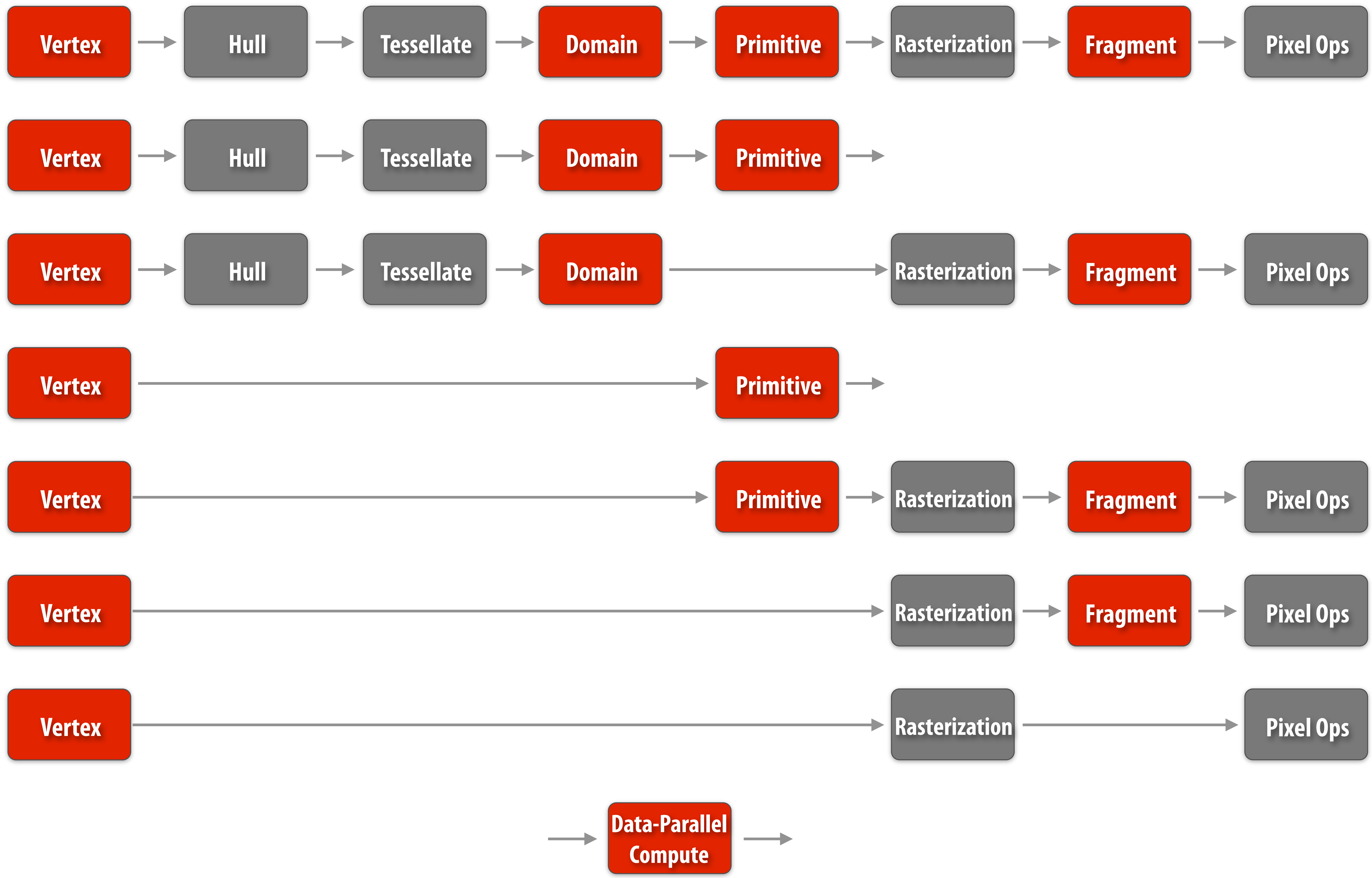
[Direct3D 11, OpenGL 4]



Added <u>three</u> new stages (new data flows needed to support high-quality surfaces)

Forked off a separate 1-stage pipeline (a.k.a. "OpenCL/CUDA)
(with relaxed data-access and communication/sync rules)

Modern graphics pipeline: highly configurable structure



Direct3D 11, OpenGL 4 pipeline configurations

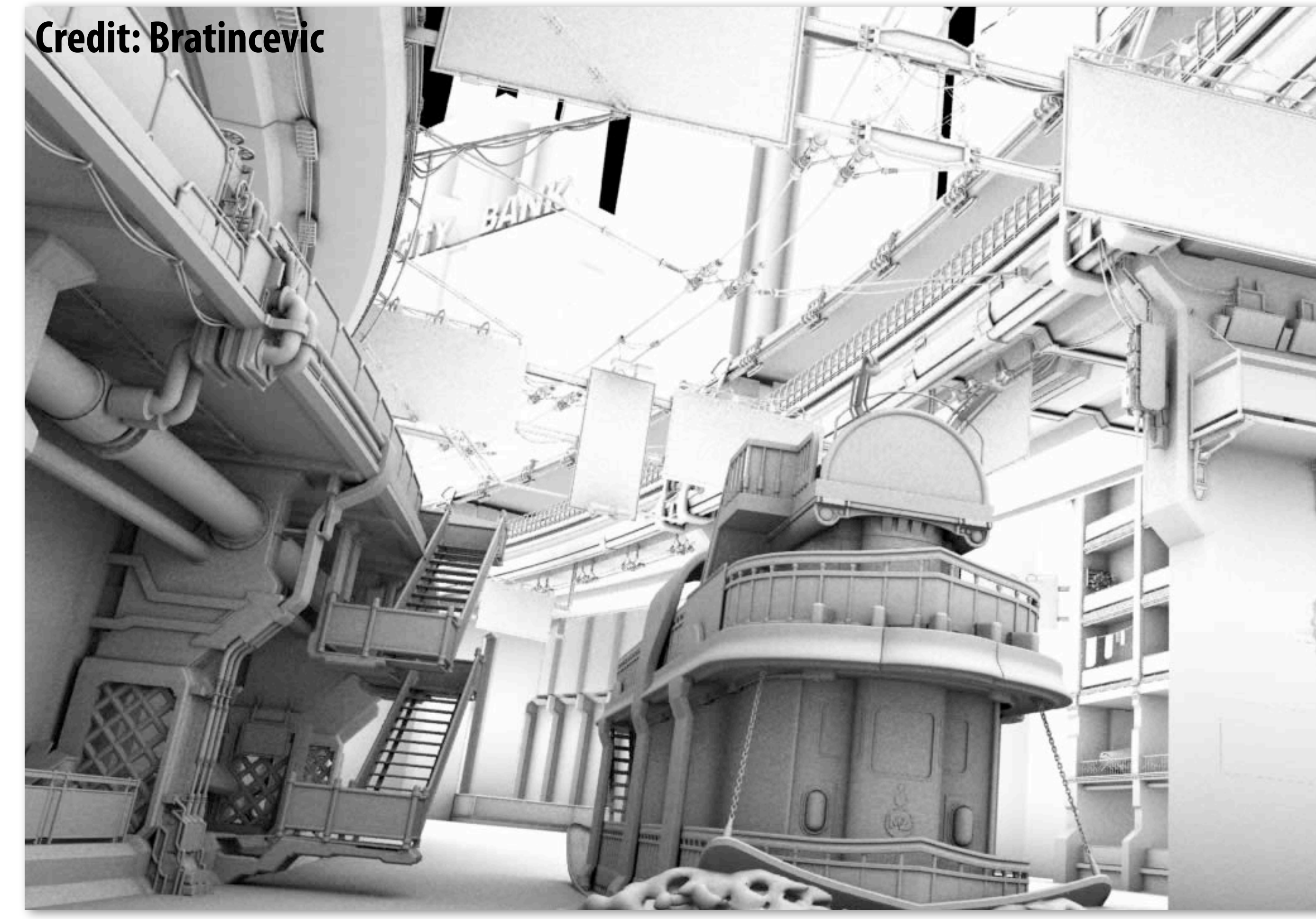
Current trends in interactive graphics

- Rapid parallel algorithm development in community
- Increasing machine performance and flexibility (e.g., heterogeneous capabilities)
  - "Traditional" discrete GPU designs
  - Most modern systems are hybrid CPU + GPU platforms

↓

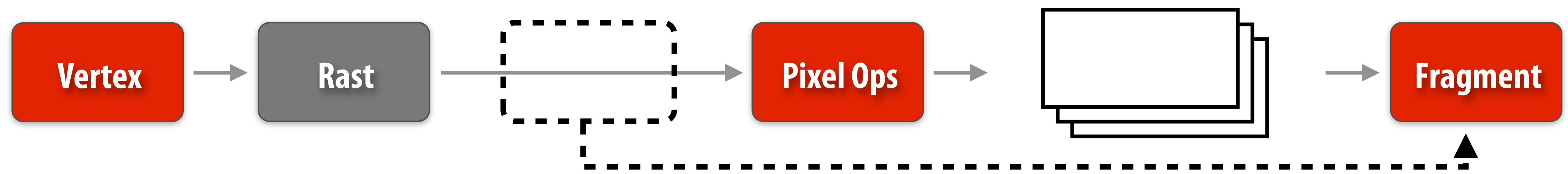
Space of candidate algorithms for future real-time use is growing rapidly

Example: global illumination algorithms



Credit: Cyril Crassin

Alternative shading structures ("deferred shading")



1000 lights, [Andersson 09]

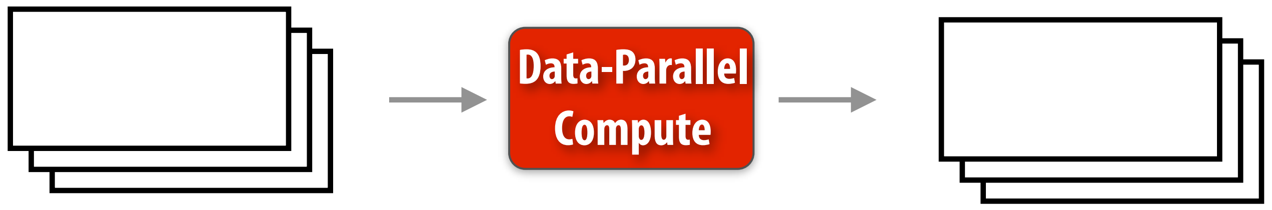
Game physics / simulation / procedural geometry

Credit: Inigo Quilez

Parallel programming model challenge

- **Future interactive systems → broad application scope**
  - Not all algorithms map elegantly to current pipeline structure
  - Pipeline structure could be extended further, but complexity is growing unmanageable
- **Must retain high efficiency typical of current systems**
  - Future hardware platforms (especially CPU+accelerator hybrids) will have the combination of resources for executing these workloads efficiently
  - Continue to leverage fixed-function processing when appropriate
  - How to abstract?

Option 1: discard pipeline structure, drop to lower-level frameworks



CUDA, OpenCL, ComputeShader, C++ /w libraries

Challenge

- Future interactive systems → broad application scope
  - Not a great fit for current pipeline structure
  - Pipeline structure could be extended further, but complexity is growing unmanageable
- Must retain high efficiency of current systems
  - Future hardware platforms (especially CPU+accelerator hybrids) will be designed to run these workloads well
  - Continue to leverage fixed-function processing when appropriate

A unique (undesirable?) property of GPU design

- **The fixed-function components on a GPU control the operation of the programmable components**
  - Fixed function logic generates work (e.g., input assembler, tessellator, rasterizer all generate elements for processing by programmable cores)
  - Programmable logic processes elements

- **In other words... application-programmable logic forms the inner loops of the rendering computation, not the outer loops!**

- **Ongoing research question: can we flip this design around?**
  - Maintain efficiency of heterogeneous hardware implementation, but give programmers control of how hardware is used and managed.

Today -- GRAMPS: one example of flipping the pipeline around

GRAMPS: A Programming Model for Graphics Pipelines

[Sugerman, Fatahalian, Boulos, Akeley, Hanrahan 2009]

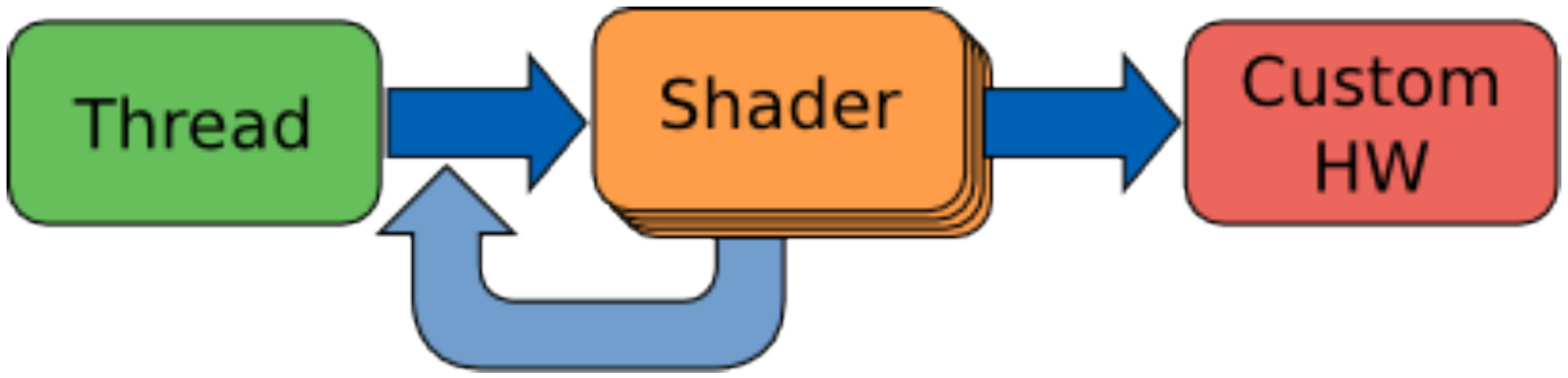
GRAMPS programming system: goals

- Enable development of application-defined graphics pipelines
  - Producer-consumer locality is important
  - Accommodate heterogeneity in workload
    - Many algorithms feature both regular data parallelism and irregular parallelism (recall: current graphics pipelines encapsulate irregularity in non-programmable parts of pipeline)

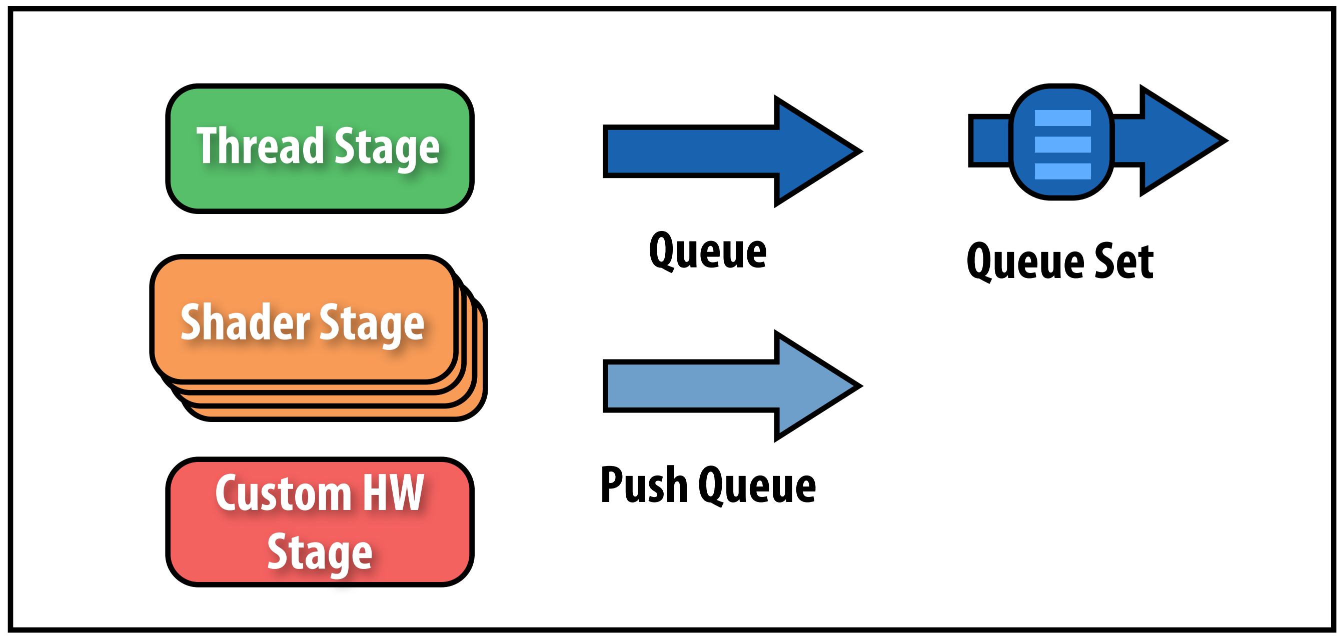
- High performance: target future CPU+GPUs (embrace heterogeneity)
  - Throughput ("accelerator") processing cores
  - Traditional CPU-like processing cores
  - Fixed-function units

GRAMPS overview

- Programs are graphs of stages and queues
  - Expose program structure
  - Leave stage internals largely unconstrained

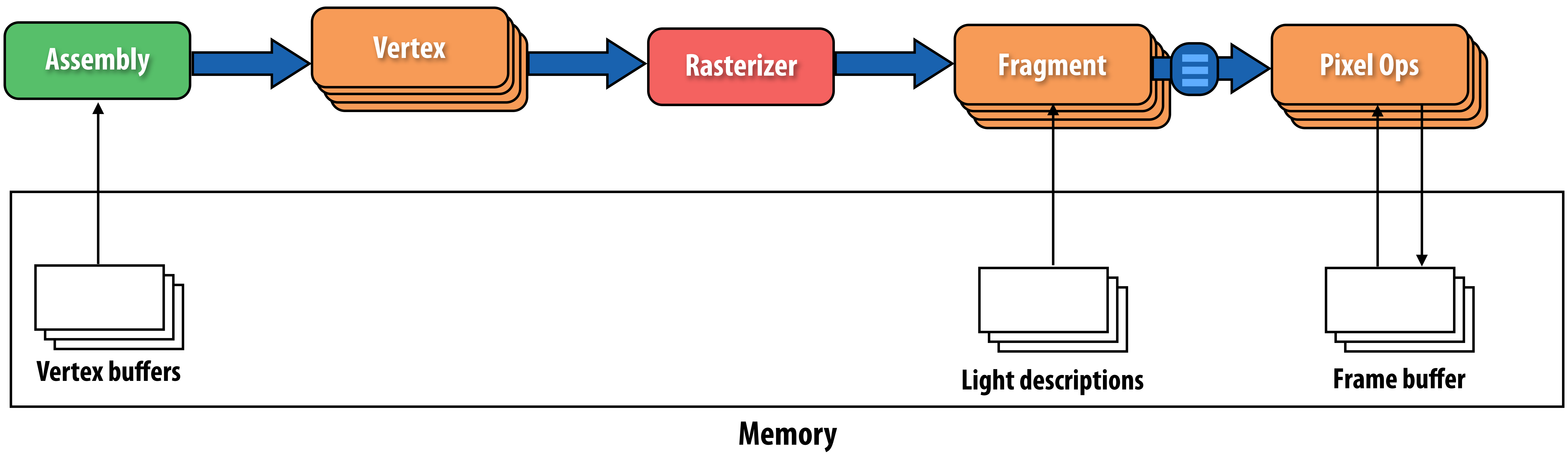


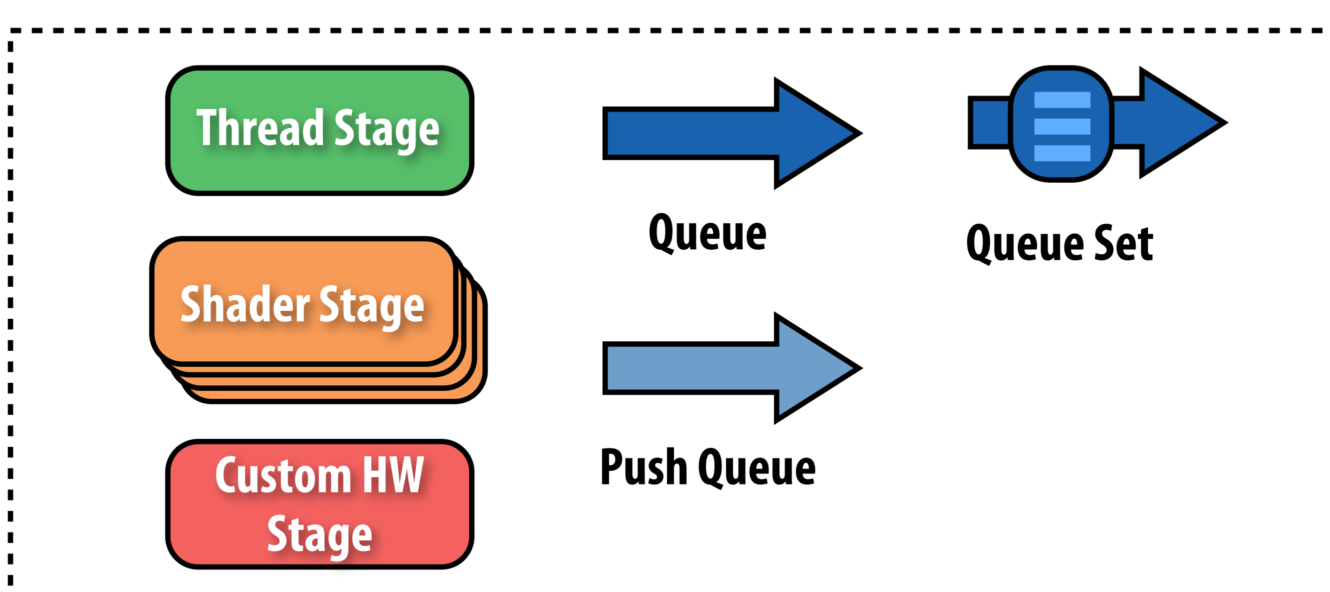
GRAMPS primitives



Writing a GRAMPS program

1. Design application graph and queues
2. Implement the stages
3. Instantiate graph and launch

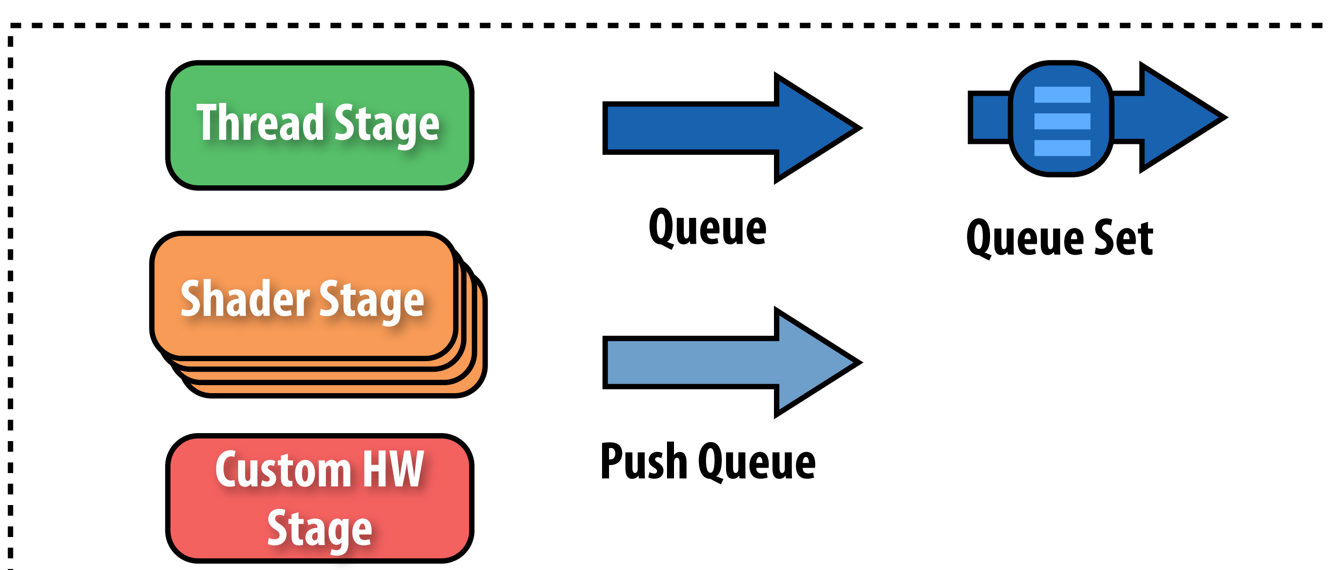




Queues

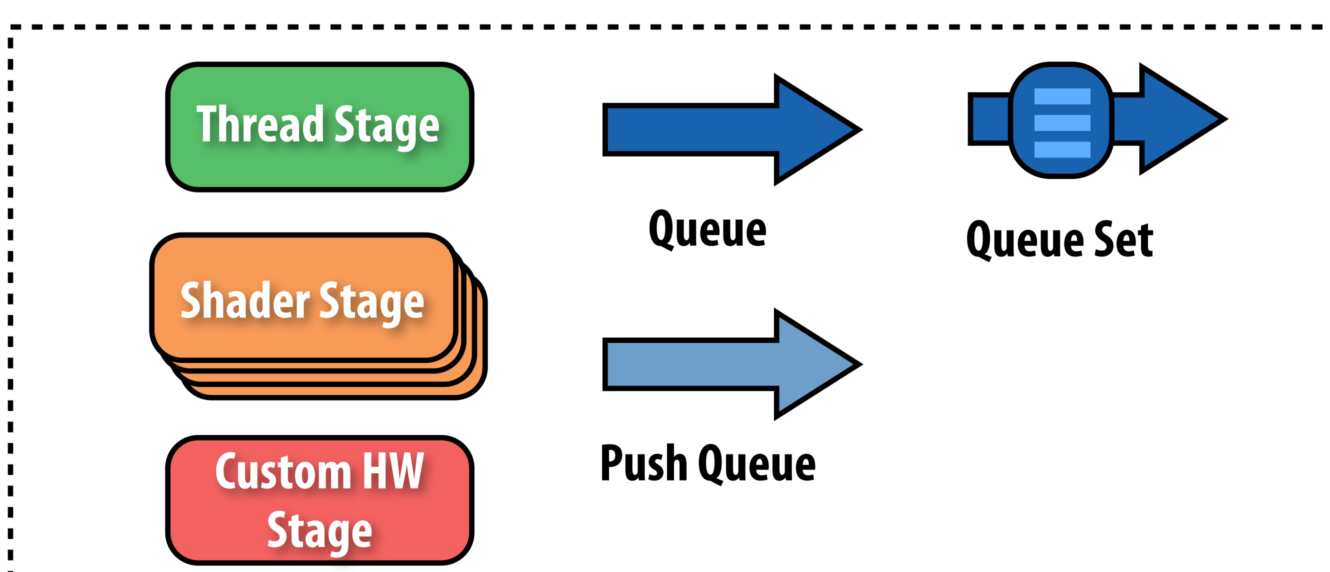
- Bounded size, operate at granularity of "packets" (structs)
  - Packets have one of two formats:
    1. Blob of data: completely opaque to system
    2. Header + array of opaque elements

- Queues can be ordered (FIFOs) or unordered FIFOs



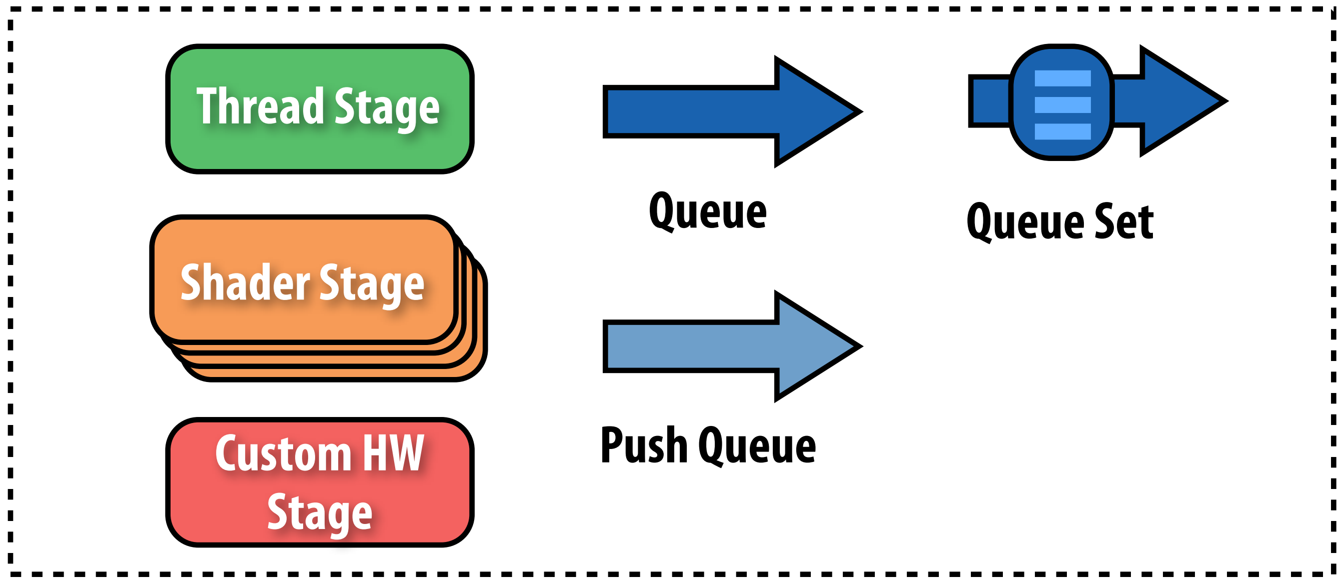
“Thread” and custom HW stages

- Preemptible, long-lived and stateful (think pthreads)
  - Threads orchestrate computation: merge, compare repack inputs
- Manipulate queues via in-place `reserve/commit`
- Custom HW stages are logically just threads, but implemented by HW



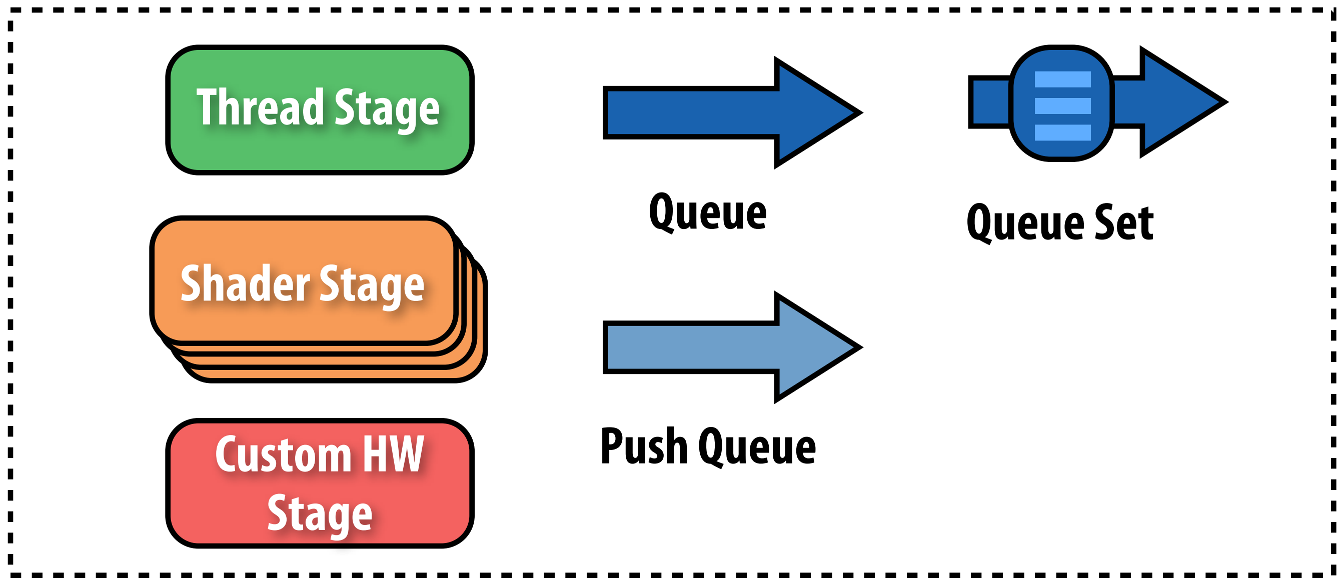
“Shader” stages

- System support for data-parallel execution
  - Logic is defined per element (like graphics shaders today)
  - Automatically instanced and parallelized by GRAMPS
- Non-preemptible and stateless
  - System has preserved queue storage for inputs/outputs
- `Push`: allows shader stage invocation to output variable number of elements to output queue
  - GRAMPS coalesces output into full packets (of header + array type)

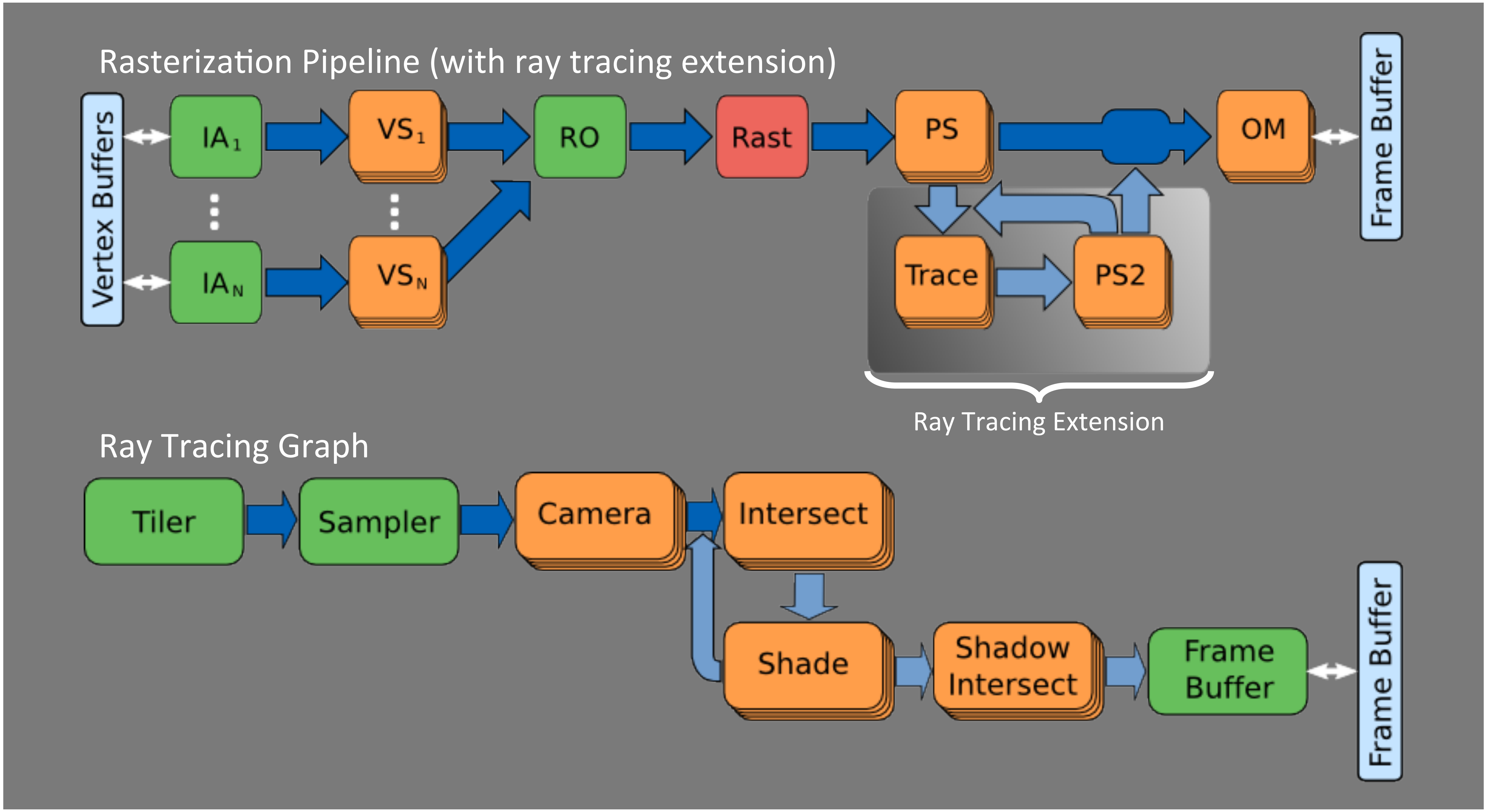


Queue sets (for mutual exclusion)

- Like N independent serial subqueues (but attached to a single instanced stage)
  - Subqueues can be created statically or "on-demand" on first output
  - Can be sparsely indexed (can think of subqueue index as a key)

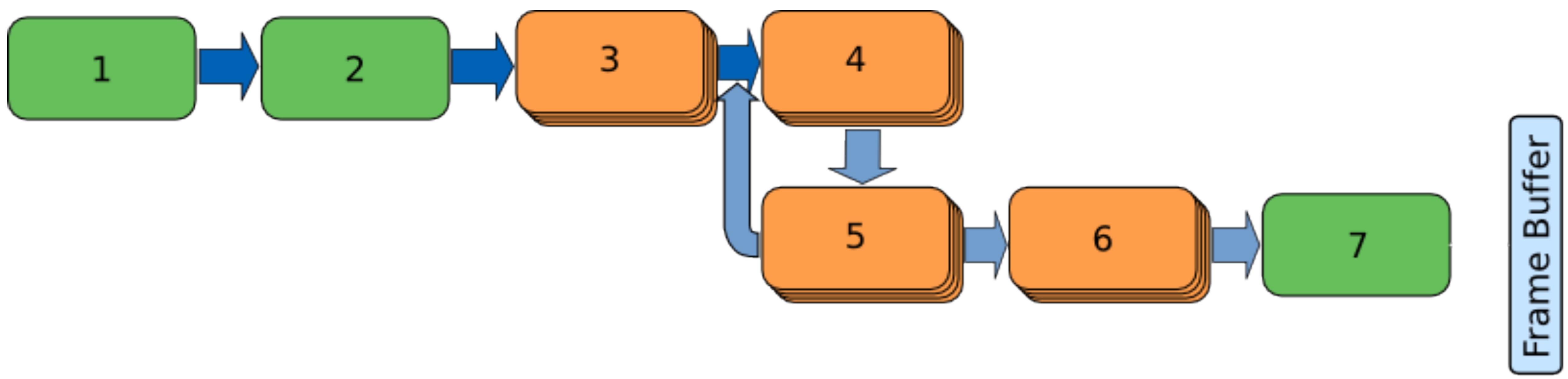


Graphics pipelines in GRAMPS



Key challenge: scheduling GRAMPS pipelines

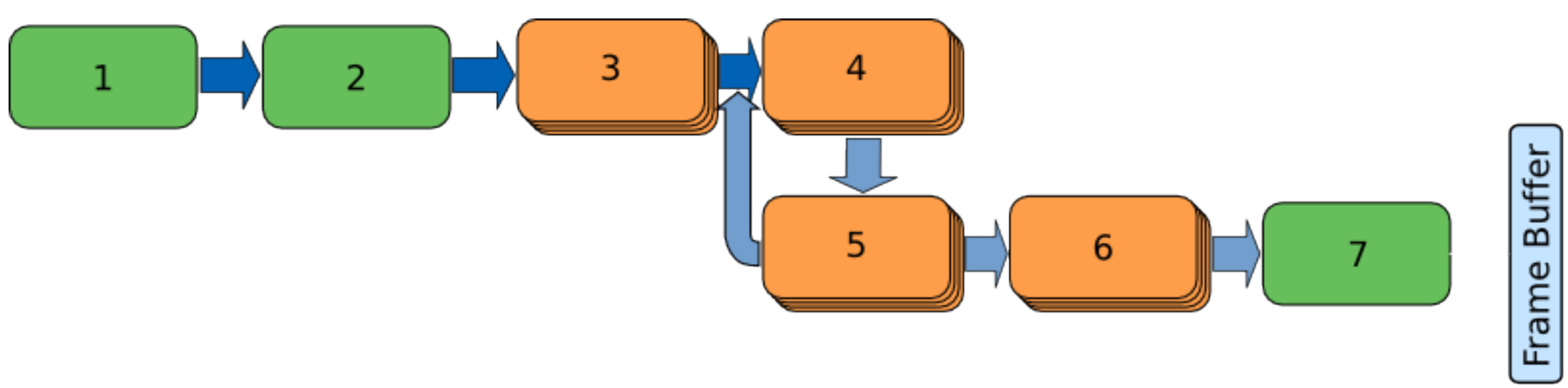
- **Naive scheduler:**
  - Use graph structure to set simple stage priorities
  - Only preempt Thread Stages on `reserve/commit` operations



Stage numbers are scheduling priorities (lowest number = highest priority)
Always execute lowest-numbered stage that has work.
Result: "breadth-first" scheduler

Key challenge: scheduling GRAMPS pipelines

- Other scheduling policies:
  - "Breadth first" always schedule lowest numbered stage with work
    - Maximizes parallelism
    - Maximizes queue lengths
    - Minimizes switching overheads
  - "Depth first" always schedule lowest priority stage with work
    - Minimizes queue lengths (produce, then immediately consume)
    - Potentially higher switching overheads due to frequent switching
  - Dynamic priorities based on queue lengths:
    - Keep queue lengths above low watermark, below high watermark



GRAMPS recap

- Key abstraction is the computation graph: typed stages and queues
  - Thread, fixed-function, and "shader" stages
  - A few types of queues: ordered, unordered, queuesets

- Key underlying ideas:
  - Enforcing structure on computations is useful for system optimization
  - Embrace heterogeneity in application and machine architecture
    - Interesting graphics applications have tightly coupled irregular parallelism and regular data parallelism (this should be encoded in structure)

- Alternative to current design of CUDA/OpenCL
  - These systems enforce very little global structure (very flexible, but provide few mechanisms for programmer to indicate intent to the system)
  - Result: these systems can only make simple mapping/scheduling decisions

GRAMPS postmortem

- Initial goal: make the graphics pipeline structure programmable

- We ended up with a lower level abstraction than today's pipeline: GRAMPS lost domain knowledge of graphics (graphics pipelines are implemented on top of GRAMPS abstractions)
  - Good: now programmable logic controls the fixed-function logic (in the current graphics pipeline it is the other way around)
  - Good: system is not graphics-domain-specific, but remains aware of program's overall structure (GRAMPS graph)

- Reality: mapping graphics abstractions to GRAMPS abstractions efficiently requires a near expert graphics programmer
  - Coming up with the right graph is hard (setting packet sizes, queue sizes has some machine dependence, some key optimizations are global)

Graphics programming abstractions today

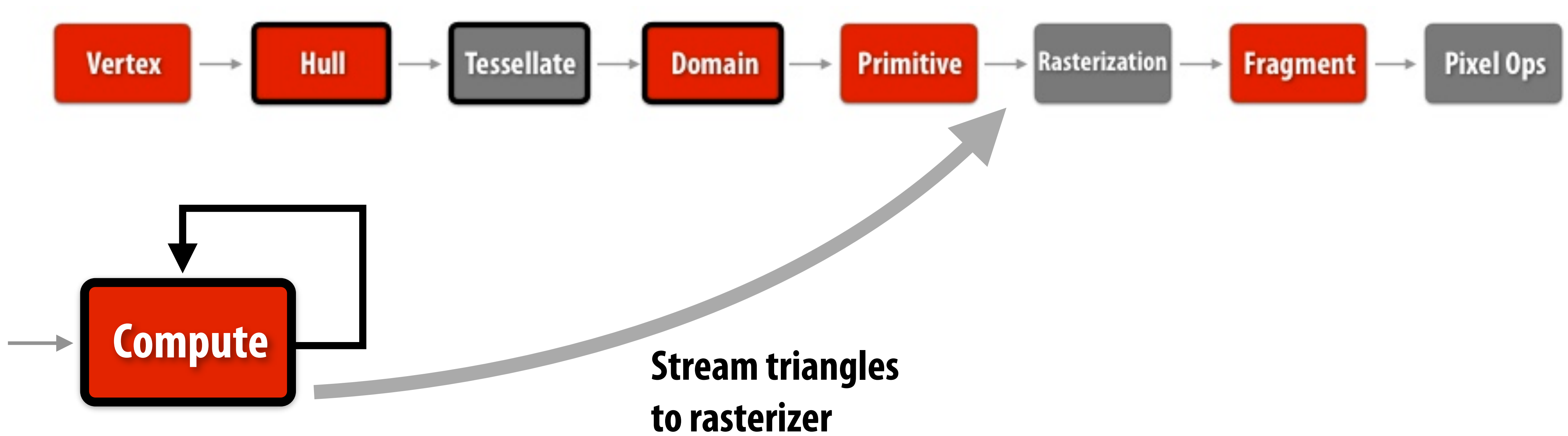
- **CPU+GPU fusion is begging for improvements to high-level frameworks for interactive graphics**
  - Example: AMD's Mantle
    - Alternative interface to AMD GPUs (few public details at this time)
  - Example: NVIDIA Optix: new framework for ray tracing
    - Application provides key kernels, Optix compiler/runtimes schedules
    - Built on top of CUDA

- **Unresolved challenge: no clear, good solution yet**
  - Echoes to broader trend in computer science: how to enable software development for parallel, heterogeneous systems
  - Mobile SoC designers are particularly interested in this problem (even more functional blocks: DSPs, camera image processors, misc sensor processors, ...)

Visual computing systems: ongoing/future systems research challenges (ideas from the course)

Visual computing: systems research challenges

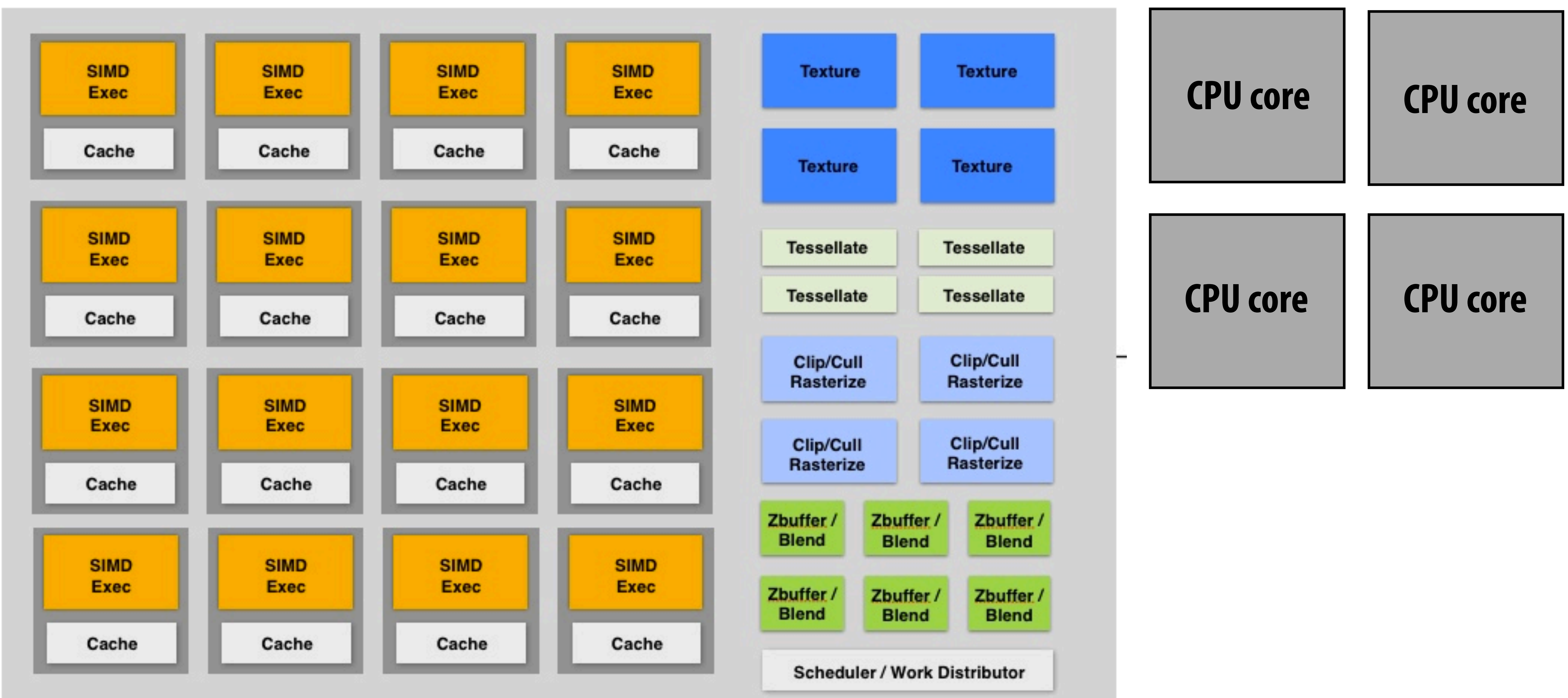
1. **Tighter integration of graphics pipeline and non-graphics pipeline workloads**
   - Many different types of computations are required to generate a frame, and not all are best carried out using the graphics pipeline
     - Geometry synthesis (tessellation, procedural geometry)
     - Parallel construction of data structures: e.g., geometry buckets, light lists, sparse voxel octree, BVH
     - Shading (data-parallel, compute intensive)
     - Image post-processing: image filtering operations such as MLAA, motion/defocus blur, tone mapping



Visual computing: systems research challenges

2. Hardware support for software-controlled fixed-function units

- What specialized hardware building blocks could be implemented to help with scheduling?



Visual computing:
systems research challenges

3. **Is there a need for distinct programmable hardware for computational photography and image understanding tasks?**
   - Or is it best to implement a few basic primitives in silicon (convolution, feature extraction, histogram generation, etc.)
   - And then rely on GPU-like throughput processors for programmability

Visual computing: systems research challenges

3. **Unique rendering challenges for virtual reality**
   - (Sadly, left out of this course) see Michael Abrash's GDC Keynote

4. **New abstractions/architectures for analyzing images and video at scale**
   - Content-based retrieval as a key computational primitive